8K

УДК 519.686.2

А.В. Колчин
Институт кибернетики имени В.М. Глушкова НАН Украины, г. Киев, Украина
kolchin_av@yahoo.com

Автоматический метод оперативного построения абстракций при верификации формальных моделей асинхронных систем

Предложен метод построения точных абстракций «на лету» и его использование в верификации формальных моделей. Метод основан на том, что каждое пройденное состояние модели характеризуется неполным набором атрибутов, при этом достигается существенное сокращение числа состояний, необходимых для анализа верифицируемой модели. Описаны основные алгоритмы построения абстракций, приведены примеры, иллюстрирующие эффективность применения, а также необходимые расширения для проверки темпоральных свойств.

Введение

С возрастанием сложности программных систем обостряется актуальность автоматизации проверки их правильности. Одним из основных методов автоматической верификации есть проверка модели формальных спецификаций систем (model checking [1]). В настоящее время наиболее популярны такие верификаторы, как SPIN [2], [3], SMV [4], NuSMV [5], Verilog [6]. Основная проблема верификации есть проблема комбинаторного взрыва состояний модели. Решению проблемы посвящено множество различных методов – накладываются ограничения на пространство поиска [7], используются различные абстракции и аппроксимации [8-11], методы частичного порядка [12], [13], использование симметрий при проверке эквивалентности состояний [14], [15], исследование зависимостей [16], направленный поиск [17]. Многие верификаторы используют заданные пользователем абстракции и специальные состояния («fairness constraints», «hints», «restricted states») [18], [19] для отсечения ветвей поведения. В последнее время популярной техникой проверки моделей стала техника построения абстракций (в частности, абстракций предикатов) с последующим уточнением (т.н. abstraction-refinement). Уточнения подразумевают повторные запуски эксперимента для устранения ложных поведений. Такой подход часто называется CEGAR (Counter-Example Guided Automated Refinement – автоматическое уточнение, основанное на анализе контр-примеров), и составляет основу для многих популярных верификаторов [6], [8], [9], [11], [20].

Данная работа описывает метод автоматического построения абстракций «на лету», в основе которого лежит анализ истории изменения и использования значений атрибутов. Так, для каждого пройденного состояния выделяется достаточное для проверки свойств модели подмножество атрибутов. Ниже даны формальные определения моделей и проверяемых свойств, после чего описаны алгоритмы и приведены их обоснования. В завершение представлены примеры и сравнительный анализ с существующими методами и некоторыми популярными верификаторами, а также необходимые расширения метода для проверки темпоральных свойств.

1. Формальные модели и их свойства

Операционная семантика модели может быть определена в терминах транзиционных систем (ТС). ТС – это тройка $T = (Q, q_0, \rightarrow)$, где Q – множество состояний, $q_0 \in Q$ – начальное состояние, $\rightarrow \subseteq Q \times Q$ – конечное множество детерминированных переходов, записывается $q \rightarrow q'$, если существует переход из состояния q в q', такой переход будем называть допустимым из состояния q. Путь $path$ из состояния q_0 в q_k есть (конечная) последовательность состояний $q_0 \rightarrow q_1 \rightarrow ... \rightarrow q_k$. Состояние q достижимо, если существует путь из q_0 в q. Будем обозначать $\delta(q)$ все достижимые из q состояния. Размеченная ТС обозначается $TL = (T, L)$ где T – ТС, в которой множество переходов размечено: $\rightarrow = \bigcup_{t \in L} \xrightarrow{t}$. Используются реализации атрибутных ТС (АТС) вида $M = (TL, A, D, I, F)$, где A обозначает конечное множество атрибутов, $D(v)$ есть конечная область значений для каждого атрибута $v \in A$; TL – размеченная ТС с конечным числом состояний, у которой переходы $\rightarrow = \bigcup_{t \in L} \alpha \xrightarrow{t} \beta$ имеют α – предусловие в виде бескванторной формулы (классической) логики предикатов, t – имя перехода, β – постусловие, представляющее собой набор присваиваний вида $v := F(Q, A)$; I – интерпретация атомарных формул, а функция F – интерпретация правых частей присваиваний в постусловиях. В описании алгоритмов будем использовать функцию `evaluate` как для I, так и для F.

Определение 1. *Конкретной* АТС есть $M_c = (TL_c, A, D, I, F)$, в которой каждое состояние системы $q_c \in Q_c$ характеризуется множеством значений всех атрибутов: $q_c = \{\bigcup_{0 \le i \le |A|} (v_i = d_i) | d_i \in D(v_i)\}$.

Определение 2. В *абстрактной* АТС $M_a = (TL_a, A, D, I, F)$ для каждого состояния системы $q_a \in Q_a$ существует подмножество $A_a \subseteq A$, что $q_a = \{\bigcup_i (v_i = d_i) \mid v_i \in A_a\}$.

Определение 3. Истинность формулы φ на состоянии пути p_i, записывается $p_i|=\varphi$, определяется индуктивно по структуре формулы φ:

$p_i \models a \equiv I(p_i, a) = T$ – *если атомарная формула a истинна в i-м состоянии*;

$p_i \models \neg\varphi \equiv p_i \not\models \varphi$ – *если формула φ не выполняется в p_i*;

$p_i \models \varphi_1 \wedge \varphi_2 \equiv p_i \models \varphi_1 \wedge p_i \models \varphi_2$ – *если в p_i выполняется формула φ_1 и φ_2*;

$p_i \models \varphi_1 \vee \varphi_2 \equiv p_i \models \varphi_1 \vee p_i \models \varphi_2$ – *если в p_i выполняется формула φ_1 или φ_2*;

Определение 4. Переход t достижим, если существует достижимое состояние $q \in Q$, такое, что $q \xrightarrow{t} q' \mid q' \in Q$.

Определение 5. Назовем тупиковым состояние q^d, из которого не существует ни одного допустимого перехода: $\forall t \in \rightarrow : q^d \not\models \alpha_t$.

Определение 6. Назовем q^l состоянием ливлока, если $q^l \in \delta(q_0) \wedge \delta(q_0) - \delta(q^l) \neq \varnothing$.

Определение 7. Назовем недетермированным состояние q^n, для которого существует более чем один допустимый переход: $\exists t_1, t_2 \in \rightarrow : q^n \models \alpha_{t_1} \wedge q^n \models \alpha_{t_2}$.

Отметим, что если тупиковая ситуация (deadlock) почти всегда интерпретируется как ошибка, то недетерминизм (transition inconsistency) и ливлок (livelock) в модели – скорее, сигнал предупреждения для дальнейшего анализа.

Определение 8. Назовем абстрактную АТС M_a точной абстракцией конкреетной АТС M_c по отношению к проверяемым свойствам Ψ, если выполняется $M_a \models \Psi \Leftrightarrow M_c \models \Psi$.

Цель данной работы состоит в автоматическом построении точных абстракций сос-тояний при проверке таких свойств системы, как достижимость переходов, тупиков, ливлоков и недетерминизмов, а также свойств, заданных пользователем.

2. Описание метода

Интуитивно, суть метода заключается в игнорировании некоторых незначимых значений атрибутов. Под «незначимым» на некотором состоянии S понимается *значение* атрибута, которое не используется ни одним переходом, а также не различается ни одним из проверяемых свойств на всех состояниях, достижимых из S. Таким образом, незначимые значения можно не запоминать, и тем самым ослабить проверку эквивалентности состояний. Соответственно, «значимым» на состоянии есть значение атрибута, которое используется либо для определения допустимости перехода, либо для проверки свойств. Заметим, что существование состояний в модели, содержащих такие незначимые значения атрибутов, не всегда означает наличие избыточного присваивания в постусловии. Такие присваивания в одной ситуации могут быть значимыми, а в другой – нет. Не теряя общности, предполагается, что ни предусловия, ни постусловия: не допускают деления на 0, выходов за пределы допустимых значений и использования неинициализированных атрибутов. Анализ таких свойств потребует введения соответствующих вспомогательных атрибутов и проверок; требуемая модификация [21] носит чисто технический характер и не является критичной для доказательства основных свойств алгоритмов. Рассмотрим пример некоторого предусловия: $X \wedge Y \vee Z$. Непосредственно из таблицы истинности формулы следует, что если значение атомарной формулы X ложно, то значение Y не влияет на результат, т.е. значение Y может быть любым; аналогично, значение Z можно не вычислять, если X и Y истинны. На практике, во многих языках программирования, таких как Си [22] и пр., бинарные логические операции, вообще говоря, интерпретируются как некоммутативные и, как правило, ассоциируются слева направо. Сначала всегда вычисляется первый операнд; если его значения достаточно для определения результата операции, то второй операнд не вычисляется. Приведем пример:

```
if(p != NULL && p->attr == 0)…
```

если поменять местами конъюнкты, то в результате выполнения этого участка программы со значением `p = NULL` произойдет сбой сегментации. Далее описан алгоритм, преобразующий логическую формулу в программу, интерпретирующую предусловия переходов с учетом некоммутативности логических операторов.

Отметим, что из соображений простоты описания и доказательств, приведенные алгоритмы не претендуют на оптимальность.

Алгоритм 1. Интерпретация предусловий.

Вход. Имя перехода `t` и формула предусловия `Pre`.

Выход. Процедура `precond[t]`, интерпретирующая предусловие `Pre`.

```
interpret_pre := proc(t, Pre)begin
   return ` precond[t] := proc(S)local(R_SET, result) begin
                R_SET ← ∅;' !
                truth_table(Pre, T) !
                `return (result; R_SET) end'
end
```

Оператор truth_table представляет собой систему переписывающих правил, которая сопоставляет входной терм левой части (до знака «=») каждого правила сверху вниз до первого подходящего, и на выходе строит терм согласно правой части правила; **T**, **F** обозначают True, False; evaluate вычисляет значение атомарных формул:

```
truth_table := rewrite_system(a, b)begin
1. (a \/ b, T) = ‘begin ’ ! truth_table(a,T) ! ‘end
                    if(result=F)then do begin ’ ! truth_table(b,T) ! ‘end’,
2. (a \/ b, F) = ‘begin ’ ! truth_table(a,F) ! ‘end
                    if(result=T)then do begin ’ ! truth_table(b,F) ! ‘end’,
3. (a & b, T)  = ‘begin ’ ! truth_table(a,T) ! ‘end
                    if(result=T)then do begin ’ ! truth_table(b,T) ! ‘end’,
4. (a & b, F)  = ‘begin ’ ! truth_table(a,F) ! ‘end
                    if(result=F)then do begin ’ ! truth_table(b,F) ! ‘end’,
5. (~(a), T)   = truth_table(a, F),
6. (~(a), F)   = truth_table(a, T),
7. (a, T)      = read(a) ! ‘result ← evaluate (S, ‘ ! a ! ’)’,
8. (a, F)      = read(a) ! ‘result ← ~(evaluate(S, ‘ ! a ! ’))’
end
```

Знак «!» означает операцию конкатенации. Процедура read строит операторы для добавления атрибутов, входящих в формулу, в некоторое множество R_SET:

```
read := proc(atomic_formula)local(r)begin
   r ← ‘’;
   for_each attr ∈ atomic_formula do
      r ← r ! ‘ R_SET ← R_SET ∪ ’ ! attr ! ‘;’;
   return r
end
```
□

Алгоритм 1 применяется так же для интерпретации пользовательских свойств модели. При этом вместо имени перехода первым параметром указывается идентификатор свойства.

Вход:interpret_pre(t1, (a>0 \/ b=1))
Выход:

```
precond[t1] := proc(S)local
(R_SET, result)begin R_SET ← ∅;
   R_SET ← R_SET ∪ a;
   result ← evaluate(S, a>0);
   if (result = F) then do begin
      R_SET ← R_SET ∪ b;
      result ← evaluate(S, b=1)
   end
   return (result; R_SET)
end
```

Вход:interpret_pre(safety1, ~(a>0 & b=1))
Выход:

```
precond[safety1] := proc(S)local
(R_SET, result)begin R_SET ← ∅;
   R_SET ← R_SET ∪ a;
   result ← ~(evaluate(S, a>0));
   if (result = F) then do begin
      R_SET ← R_SET ∪ b;
      result ← ~(evaluate(S, b=1))
   end
   return (result; R_SET)
end
```

Рисунок 1 – Пример работы алгоритма 1

Лемма 1. Алгоритм 1 строит процедуру precond, которая:

а) интерпретирует логическую формулу согласно таблицам истинности для некоммутативных логических выражений, содержащих только пропозициональные связки и имеет на выходе **T**, если формула истинна, и **F** в противном случае;

б) для бинарных операций не вычисляет второй операнд, если значения первого операнда достаточно для определения результата операции;

в) множество R_SET на выходе содержит только атрибуты, входящие в операнды, значения которых вычислялись.

Доказательство:

а) преобразование пропозициональных связок (truth_table, правила 1 – 6) и вычисление результата интерпретации атомарных формул, т.е. вызов функции evaluate (правила 7 – 8) производится с учетом полярности (**T**, **F**) согласно таблицам истинности для некоммутативных логических выражений;

б) согласно семантике операторов if-then, вычисление второго операнда осуществляется только в случае, когда значения первого операнда не достаточно для определения результата операции (truth_table, правила 1 – 4);

в) базис рекурсии (последние два правила) содержит вызов процедуры evaluate, которая вычисляет результат атомарного выражения. А так как процедура read вызывается только перед вызовом evaluate с тем же параметром, и только она формирует множество R_SET, то множество будет содержать только атрибуты, входящие в операнды, значения которых вычислялись.

Следствие 1. Значений атрибутов множества R_SET на выходе процедуры precond достаточно для определения допустимости перехода.

Процедура property_check формирует множество атрибутов R_SET, которые используются для проверки всех пользовательских свойств модели на состоянии S:

```
property_check := proc(S, Ψ)local(RESULT, R_SET, res, r)begin
   RESULT ← F; R_SET ← ∅;
   for_each p ∈ Ψ do begin
      (res; r) ← precond[p](S);
      if (res = T) then do RESULT ← T;
      R_SET ← R_SET ∪ r
   end
   return (RESULT; R_SET)
end
```

Следствие 2. Значений атрибутов множества R_SET на выходе процедуры proper-ty_check достаточно для определения истинности пользовательских свойств.

Заметим, что R_SET будет содержать не обязательно все атрибуты, входящие в предусловие перехода или формулу свойства модели. Для краткости, будем называть атрибуты множеств R_SET R-атрибутами.

Рассмотрим далее алгоритм 2, который будет использоваться для построения процедур для интерпретации постусловий. Из соображений простоты описания предполагается, что присваивания в постусловии коммутативны, т.е. не содержат повторных присваиваний и не используют новых значений атрибутов.

Алгоритм 2. Интерпретация постусловий.

Вход. Переход t, множество присваиваний постусловия Post, так же исходное состояние Src, и целевое Dst.

Выход. Процедура postcond[t] для интерпретации присваиваний Post, а так же множества W_SET и V_SET[].

```
interpret_post := proc(t, Post)local(r, w, expr, v)begin
  r ← 'postcond[t] := proc(Src, Dst)local(W_SET,V_SET[])begin
                          W_SET ← ∅; V_SET[ ] ← ∅;' ;
  for_each (w := expr) ∈ Post do begin
     r ← r ! 'W_SET ← W_SET ∪ ' ! w ! ';' ;
     for_each v ∈ expr do
        r ← r ! 'V_SET[' ! w ! '] ← V_SET[' ! w ! '] ∪ ' ! v ! ';' ;
     r ← r ! 'Dst->' ! w ! ' ← evaluate(Src, expr)' ! ';'
  end
  r ← r ! 'return (Dst; W_SET; V_SET) end' ;
  return r
end
```

Вход:
```
interpret_post(t1, (a := a - 1; c := b))
```
Выход:
```
postcond[t1] := proc(Src, Dst)local(W_SET, V_SET[])begin
   W_SET ← ∅; V_SET[] ← ∅;
   W_SET ← W_SET ∪ a;   V_SET[a] ← V_SET[a] ∪ a;
   Dst->a ← evaluate(Src, a - 1);
   W_SET ← W_SET ∪ c;   V_SET[c] ← V_SET[c] ∪ b;
   Dst->c ← evaluate(Src, b);
   return (Dst, W_SET, V_SET)
end
```
Рисунок 2 – Пример работы алгоритма 2

Лемма 2. Алгоритм 2 строит процедуру postcond, которая:

а) выполнит все присваивания постусловия из состояния Src в Dst;

б) формирует множество атрибутов W_SET, которым осуществлялось присваивание;

в) для каждого атрибута w из множества W_SET формирует множество V_SET[w], содержащее атрибуты, входящие в выражения, формирующие значение перезаписываемого атрибута w.

Доказательство:

а) строка 2 предполагает выполнение строки 6 для каждого присваивания; в строке 6 присваивание значения, вычисляемого из предыдущего состояния Src, выполняется для атрибута нового состояния Dst;

б) строка 2 предполагает выполнение строки 3 для каждого присваивания; строка 3 выполняет формирование множества W_SET, добавляя перезаписываемый атрибут;

в) строка 2 выполнит строку 4 для каждого присваивания; строка 4 предполагает выполнение добавления (строка 5) каждого атрибута, входящего в выражение, формирующее значение перезаписываемого атрибута w во множество V_SET[w].

Для краткости будем называть атрибуты множества W_SET W-атрибутами, а V_SET[] – V-атрибутами. Множества R-атрибутов и V-атрибутов различаются намеренно потому, что значения W-атрибутов могут не использоваться в будущем, и таким образом, соответствующие значения V-атрибутов так же останутся неиспользованными, а значит, «незначимыми». Однако, если же W-атрибут (на продолжении пути) будет участвовать в определении поведения или в проверке свойств, т.е. станет R-атрибутом, то элементы, формировавшие значение такого W-атрибута (V-атрибуты) так же станут «значимыми». Далее зададим функцию перехода transit, которая: формирует множество R-атрибутов, и, если переход допустим, новое состояние (Dst), а так же множества V- и W-атрибутов.

```
transit := proc(Transition, Src, Dst)local(RESULT,R_SET,W_SET,V_SET)begin
   (RESULT; R_SET) ← precond[Transition](Src);
   if(RESULT = T) then do
      (Dst; W_SET; V_SET) ← postcond[Transition](Src, Dst);
   return (RESULT; Dst; R_SET; W_SET; V_SET)
end
```

Предусловие перехода t1 : (a>0 \/ b=1)
Постусловие перехода t1 : (a := a - 1; c := b)
Вход:
```
transit(t1,(a = 2, b = 0), (a = 2, b = 0))
```
Выход:
```
(T;(a = 1, b = 0, c = 0);(a,c);([a]a,[c]b))
```
Рисунок 3 – Пример работы процедуры transit

Легко видеть, что согласно леммам 1 и 2, процедура `transit` удовлетворяет определению перехода АТС. В дальнейшем будем использовать структуру `trace`:

```
trace[index] = (HIST, V_SET[], W_SET, explored, deadlock, link)
```

Индекс структуры `index` используется для синхронизации со структурой `path`. Множество `HIST` необходимо для хранения истории значений атрибутов, его определение приводится ниже. Множества `V_SET[]` и `W_SET` формируются из соответствующих множеств при осуществлении перехода процедурой `transit`. Множество `explored` используется для запоминания исследованных переходов, `deadlock` как признак тупиковой ситуации, а так же связь `link` – индекс пути для нахождения сильно связных компонент графа поведения модели.

Очевидно, что для того, чтобы вычислить значение атрибута на предыдущем состоянии пути в случае, когда постусловие последнего перехода содержит присваивание этому атрибуту, достаточно знать значения всех атрибутов, находящихся в правой части такого присваивания. Очевидно также, что если постусловие последнего на пути перехода не содержит присваивания атрибуту, то его значение не изменится.

Определение 9. Множество H_a^j для атрибута a на j-м состоянии пути ($j > 0$) включает сам атрибут a, или, если его значение перезаписывалось в постусловии последнего перехода, то все атрибуты, содержащиеся в правой части соответствующего присваивания.

Очевидно, значения атрибутов множества H_a^j на состоянии q_{j-1} однозначно определяют значение атрибута a в состоянии q_j.

Определение 10. Историей значения атрибута a в i-м состоянии пути ($0 \le j < i$) есть множество $HIST_a^j = \bigcup_{h \in HIST_a^{j+1}} H_h^{j+1}$; $HIST_a^i = a$.

Очевидно, значений атрибутов множества $HIST_a^j$ в j-м состоянии достаточно для однозначного определения значения атрибута a в i-м состоянии пути.

Для формирования истории значений атрибутов будет использоваться алгоритм 3, который к (ранее сформированным) множествам `trace[j]->HIST` добавляет историю атрибута a в каждое состояние пути $0 \le j \le i$.

Алгоритм 3. Формирование истории значений атрибутов.
Вход. Структура `trace` (с множествами `HIST`, `W_SET`, `V_SET` на каждом состоянии), длина пути `i`, атрибут `a`.
Выход. Дополненная историей атрибута `a` структура `trace`.

```
  form_hist := proc(trace, j, i, a)local(r, h)begin
1.   r ← ∅;
2.   if(j = i) then do r ← a
     else do begin
3.      hist = form_hist(trace, j + 1, i, a);
4.      for_each h ∈ hist do r ← r ∪ H(trace, j + 1, h)
     end
5.   trace[j]->HIST ← trace[j]->HIST ∪ r;
6.   return r
  end
  H := proc(trace, j, a)begin
     if(a ∈ trace[j-1]->W_SET) then do r ← trace[j-1]->V_SET[a]
     else do r ← a;
     return r
  end                                                  □
```

```
Вход:
trace=(
trace[0]:HIST=cf; W_SET=cf,z; V_SET:[z]=z
trace[1]:HIST=cf; W_SET=cf,b; V_SET:[b]=z ), j=0, i=1, a=b}
```

Вход: (добавлены атрибуты z и b)

```
trace=(
trace[0]:HIST=cf,z; W_SET=cf,z; V_SET:[z]=z
trace[1]:HIST=cf,b; W_SET=cf,b; V_SET:[b]=z  )
```

Рисунок 4 – Пример работы алгоритма 3

Лемма 3. Алгоритм 3 правильно формирует истории значений атрибутов.

Доказательство. Легко видеть, что процедура `H` правильно вычисляет соответствующую функцию H_h^j. Строка 2 процедуры `form_hist` обрабатывает базис рекурсии, записывая в результирующую переменную `r` атрибут `a`. Строка 3 реализует, согласно определению 10, рекурсивный вызов для вычисления истории на следующем состоянии. Строка 4 обеспечивает вызов функции `H` для каждого элемента истории следующего состояния и объединение результатов в переменную `r`. Строка 5 выполняет объединение ранее заполненной структуры и построенного множества истории для атрибута `a`. Заметим, рекурсия увеличивает второй параметр, и согласно строкам 2 и 6 при достижении значения, равному длины пути, алгоритм остановится.

Далее опишем основной алгоритм проверки свойств модели, в основе которого лежит алгоритм Тарьяна [23]. Алгоритм реализует обход пространства поведения модели, а также находит все компоненты сильной связности. Последнее свойство используется для обнаружения ливлоков, а также для проверки темпоральных свойств.

Алгоритм 4. Проверка свойств модели.

Вход. Атрибутная транзиционная система (в которой множества состояний еще не построены), начальное состояние q_0 и свойства Ψ.

Выход. Множество трасс, ведущих к нарушению свойств, и множество абстрактных состояний `visited`.

```
model_check := proc()begin
   visited ← ∅; stack ← ∅;
   initialize_state(0);
   traverse(0)
end
```

Множество `visited` будет содержать достижимые абстрактные состояния, `stack` – состояния компонент сильной связности. Для инициализации состояний используется процедура `initialize_state`:

```
initialize_state := proc(i)begin
  if i = 0 then do path[0] ← q0
  else do path[i] ← path[i-1];
  trace[i]->HIST ← ∅; trace[i]->V_SET ← ∅; trace[i]->W_SET ← ∅;
  trace[i]->explored ← ∅; trace[i]->deadlock ← T; trace[i]->link ← i
end
```

Основной процедурой алгоритма есть процедура `traverse`:

```
   traverse := proc(i)local(R_T,R_C,R_V,R_P,W,V,t,res,a,link,j,x)begin
      for_each t ∈ Transitions\trace[i]->explored do begin
         trace[i]->explored ← trace[i]->explored ∪ t;
         initialize_state(i+1);
         затолкнуть path[i] в stack;
```

```
 5.       (res, path[i+1], R_T, W, V) ← transit(t, path[i], path[i+1]);
 6.       for_each a ∈ R_T do form_hist(trace, 0, i, a);
 7.       if(res = T) then do begin
 8.         if(trace[i]->deadlock = F) then do put_trace(trace, "nondet")
 9.         else do trace[i]->deadlock ← F;
          // проверка цикла
10.         (res, R_C, link) ← check_cycle(path, trace, i+1);
11.         if(res = T) then do begin
12.            for_each j ∈ (link..i) do
                              trace[j]->link ← min(trace[j]->link, link);
13.            for_each a ∈ R_C do form_hist(trace, 0, i, a);
14.            return
            end
          // проверка пройденного состояния
15.         (res, R_V) ← check_visited(path[i+1]);
16.         if(res = T) then do begin
17.            for_each a ∈ R_V do form_hist(trace, 0, i, a);
18.            return
            end
          // проверка свойств
19.         (res, R_P) ← property_check(path[i+1], Ψ);
20.         for_each a ∈ R_P do form_hist(trace, 0, i, a);
21.         if(res = T) then do put_trace(trace[i], "property");
          // заполнение структуры trace и шаг рекурсии
22.         trace[i]->W_SET ← W; trace[i]->V_SET ← V;
23.         traverse(i+1)
          end
      end
24.   store_visited(path[i], trace->HIST[i]);
25.   if(trace[i]->deadlock = T) then do put_trace(trace, "deadlock");
26.   if(trace[i]->link = i) then do begin
27.      put_trace(trace, "livelock");
28.      repeat вытолкнуть x из вершины stack; print x;
29.      until x = path[i]; print "конец сильно связной компоненты";
      end
   end
```

Процедура `store_visited` используется для сохранения пройденных состояний. Заметим, что, согласно строке 24, только истории значений атрибутов сохраняются во множестве `visited`, а так как истории формируются, согласно строкам 6,13,17,20 только из `R`-атрибутов, то множество `visited` состоит из `R`-атрибутов (т.е. «значимых» атрибутов).

```
store_visited := proc(S, A_a)local(q_a, attr)begin
   q_a ← ∅;
   for_each attr ∈ A_a do q_a ← q_a ∪ (attr = evaluate(S, attr));
   visited ← visited ∪ q_a
end
```

Процедура `check_visited` проверяет принадлежность состояния `S` множеству `visited` и в случае обнаружения возвращает **T** и множество историй атрибутов ранее пройденного абстрактного состояния, иначе **F**.

```
check_visited := proc(S)local(q_a, attr)begin
  for_each q_a ∈ visited do
    if (forall (attr = value) ∈ q_a) value = evaluate(S, attr)
      then do return (T, {attributes ∈ q_a})
   return (F; ∅)
end
```

Процедура `check_cycle` проверяет наличие цикла и, в случае обнаружения, возвращает **T** и множество историй `HIST` зацикленного состояния, иначе **F**.

```
  check_cycle := proc(path, trace, i)local(j, attr)begin
1.    j ← i - 1;
2.    while j ≥ 0 do begin
3.       if (forall attr ∈ trace[j]->HIST)
4.          evaluate(path[j], attr) = evaluate(path[i], attr)
5.       then do return (T, trace[j]->HIST, j);
6.       j ← j - 1
      end
7.    return (F, ∅, i)
  end
```

□

Теорема 1. Для каждой сильно связной компоненты SC_c графа достижимости конкретной АТС алгоритм 4 найдет, и только одну, такую сильно связную компоненту $SC_a \subseteq$ `visited` графа достижимости абстрактной АТС, в которой для каждого $q_a \in SC_a$ существует такое $q_c \in SC_c$, что $q_a \subseteq q_c$ и для каждого $q_c \in SC_c$, существует такое $q_a \in SC_a$, что $q_a \subseteq q_c$.

Доказательство.

Поиск компонент сильной связности основан на поиске циклов. Покажем, что для цикла конкретного пути C_c алгоритм найдет цикл абстрактного пути $C_a \subseteq$ `visited`, в котором для каждого $q_a \in C_a$ существует такое $q_c \in C_c$, что $q_a \subseteq q_c$ и для каждого $q_c \in SC_c$, существует такое $q_a \in SC_a$, что $q_a \subseteq q_c$.

Пусть предполагаемый абстрактный цикл начинается в состоянии `j<i`. Для проверки наличия абстрактных циклов предназначена процедура `check_cycle`, которая вызывается в строке 10. Согласно строкам 5, 6, 22, 23 алгоритма, а также леммам 1, 2 и 3, значений множества `trace[j]->HIST` атрибутов на `j`-м состоянии достаточно для однозначного определения значений всех атрибутов, определяющих допустимость всех переходов цикла на всех состояниях `j..i`. Таким образом, какими бы ни были значения атрибутов множества A – `trace[i]->HIST`, в силу конечности множества состояний АТС, существует цикл C_c конкретной АТС, в котором для каждого состояния $q_c \in C_c$ существует такое абстрактное состояние $q_a = \bigcup_{\text{attr}\in\text{trace[k]->HIST}}$ (attr = evaluate(path[k], attr)), `j≤k≤i`, что $q_a \subseteq q_c$. А так как все абстрактные состояния цикла формируются только из соответствующих конкретных состояний, то для каждого $q_a \in C_a$ существует такое $q_c \in C_c$, что $q_a \subseteq q_c$.

Таким образом, согласно строке 7 процедуры `check_cycle`, ее результатом работы будет тройка (**`F`**`, ∅, i`) в случае отсутствия цикла в состоянии `i`, или, согласно строкам 3, 4, 5 тройка (**`T`**`, hist, j`), если существует такой абстрактный цикл, в котором значения атрибутов множества `hist=trace[j]->HIST` совпадают на `j`-м и `i`-м состояниях (`0≤j<i`).

Далее нужно показать, что алгоритм находит все циклы C_c. Для этого достаточно показать индукцией по числу тех вызовов процедуры `traverse`, которые завершили работу, что по окончании `traverse(i)` множество `visited` для каждого $q_c \in C_c$ ($q_c \in \delta(path[i])$) уже содержит такие состояния $q_a \in C_a$, что $q_a \subseteq q_c$.

В силу предположения индукции можно считать, что все конкретные циклы, достижимые из состояния `i`, найдены. Очевидно, цикл всегда имеет повторяемые переходы. Множество `visited` включает, согласно строкам 6, 13, 17 и процедурам `form_hist` и `store_visited`, истории всех `R`-атрибутов (достижимых из `i` переходов `R_T`, циклов `R_C` и пройденных состояний `R_V`). Это означает, что всякое состояние, в котором значения атрибутов множества `trace[i]->HIST` совпадут, не будет иметь такого потомка q_c, из которого допустим переход, входящий в необнаруженный ранее цикл (т.е. цикл, для которого не существует такого $q_a \in$ `visited`, что $q_a \subseteq q_c$). Следовательно, все циклы будут найдены.

А так как строки 10 и 12, а также процедура инициализации обеспечивают правильную обработку связи `link` согласно алгоритму Тарьяна [23], алгоритм найдет все сильно связные компоненты SC_a, более того, согласно процедурам `store_visited` (вызов в строке 24) и `check_visited` (в строке 15), нескольким конкретным компонентам может соответствовать всего одна абстрактная.

Следствие. Число состояний в каждой компоненте сильной связности абстрактной АТС не больше, чем в соответствующей компоненте конкретной АТС.

Теорема 2. Для каждой найденной алгоритмом 4 сильно связной компоненты графа достижимости абстрактной АТС $SC_a \subseteq$ `visited` существует как минимум одна сильно связная компонента SC_c графа достижимости конкретной АТС, в которой для каждого $q_a \in SC_a$, существует такое $q_c \in SC_c$, что $q_a \subseteq q_c$, и для каждого $q_c \in SC_c$, существует такое $q_a \in SC_a$, что $q_a \subseteq q_c$.

Доказательство: аналогично доказательству теоремы 1.

Следствие. Пусть Γ_c и Γ_a – множества сильно связных компонент графов достижимости соответственно конкретной и абстрактной АТС. Тогда $|\Gamma_a| \le |\Gamma_c|$.

Теорема 3. Алгоритм 4 построит абстракцию M_a конкретной АТС M_c по отношению к проверяемым свойствам Ψ, такую, что $M_a \models \Psi \Leftrightarrow M_c \models \Psi$.

Доказательство: следует из строк 19 и 20, следствия 2 леммы 1 и теорем 1, 2.

На основании всего вышеизложенного справедливы следующие **следствия*:***

1. Число достижимых абстрактных состояний не больше, чем достижимых состояний конкретной модели, т.е. $|visited| \le |\delta(q_0)|$.

2. Если конкретная модель содержит достижимое тупиковое состояние $q_c^d \in \delta(q_0)$, то алгоритм найдет тупиковое состояние, $q_a^d \subseteq q_c^d$, и, согласно строке 25, построит соответствующую трассу.

3. Обратно, алгоритм не обнаружит тупикового состояния (и не построит трассу), если конкретная модель не содержит такого достижимого тупикового состояния.

4. Если конкретная модель содержит достижимый недетерминизм $q_c^n \in \delta(q_0)$, то алгоритм найдет $q_a^n \subseteq q_c^n$, и согласно строке 8, построит соответствующую трассу.

5. Обратно, алгоритм не обнаружит недетерминизм (и не построит трассу), если конкретная модель не содержит такого достижимого недетерминизма.

6. Если конкретная модель содержит достижимый ливлок $q_c^l \in \delta(q_0)$, то алгоритм найдет $q_a^l \subseteq q_c^l$, и согласно строкам 27 и 28, построит соответствующую трассу.

7. Обратно, алгоритм не обнаружит ливлок (и не построит трассу), если конкретная модель не содержит такого достижимого ливлока.

8. Всякий переход, достижимый в конкретной модели, останется достижимым в абстрактной модели.

9. Всякий недостижимый в конкретной модели переход останется недостижимым и в абстрактной модели.

3. Примеры

Пример 1. Пусть имеется закрытый замок, у которого `N` скважин для ключей, и пусть существуют `2N` ключей – по два на каждую замочную скважину, причем один – «правильный», а второй – «неправильный». Работа замка проходит в два этапа. На первом ключи упорядоченно устанавливаются, на втором этапе их значения считываются (в том же порядке). Если очередной ключ «неправильный», то процесс считывания останавливается, при этом замок остается закрытым. Если же все ключи правильные, замок открывается. При этом требуется проверить свойство, что если хотя бы один из установленных ключей неправильный, то замок останется закрытым. Ниже приведены спецификации на языках Promela для SPIN[24] и SMV для Cadence SMV[25], NuSMV[26], Verilog RTL для VCEGAR[27]:

SPIN:

```
byte scan = 1, next_key = 1;
bit key1 = 0; ... bit keyN = 0;
… select:
    do
      :: next_key == 1 -> {next_key = next_key + 1; key1 = 1; goto select};
      :: next_key == 1 -> {next_key = next_key + 1; key1 = 0; goto select}; ...
      :: next_key == N -> {next_key = next_key + 1; keyN = 1; goto scanner};
      :: next_key == N -> {next_key = next_key + 1; keyN = 0; goto scanner};
    od;
scanner:
    do
      :: (scan == 1 && key1 == 1) -> {scan = scan + 1; goto scanner};
      :: (scan == 1 && key1 == 0) -> {scan = 0; goto end}; ...
      :: (scan == N && keyN == 1) -> {scan = scan + 1; goto end};
      :: (scan == N && keyN == 0) -> {scan = 0; goto end};
    od; …
Свойство: never(scan == N + 1 && (key1 == 0 || … || keyN == 0))
```

Cadence SMV, NuSMV:

```
#define N N
VAR next_key :  1..(2*N + 3); VAR scan: 1..(N + 1);
VAR key1: {1, 0, 11, 10};... VAR keyN: {1, 0, 11, 10};
init(next_key) := 1; init(scan) := 1; init(key1) := 0;...init(keyN) := 0;
next(next_key) := case{
    (next_key = 1 & (key1 = 10 | key1 = 11)): N + 1;
    (next_key = N + 1 & (key1 = 1 | key1 = 0)): 2; ...
    (next_key = N & (keyN = 10 | keyN = 11)): N + N;
    (next_key = N + N & (keyN = 1 | keyN = 0)): 2*N + 1;
    (next_key = 2*N + 1 & scan = N + 1): 2*N + 2;
    1:next_key;
};
next(key1) := case{
```

```
   (next_key = 1): {10, 11};
   (next_key = N + 1 & key1 = 10): 0;
   (next_key = N + 1 & key1 = 11): 1;
   1:key1;
}; ...
next(keyN) := case{
   (next_key = N): {10, 11};
   (next_key = N + N & keyN = 10): 0;
   (next_key = N + N & keyN = 11): 1;
   1:keyN;
};
next(scan) := case{
    (next_key = 2*N + 1 & scan = 1 & key1 = 1): scan + 1;
    (next_key = 2*N + 1 & scan = 1 & key1 = 0): 0; ...
    (next_key = 2*N + 1 & scan = N & keyN = 1): scan + 1;
    1:scan;
};
```

Свойство: `!(next_key = 2*N + 2 & (key1 = 0 | … | keyN = 0))`

VCEGAR(1, 2):

```
module main (k01, k02,... kN, clock);
input clock; input [1:0] k01; ... input [1:0] kN;
reg [1:0] key01; ... reg [1:0] keyN; reg [5:0] scan; reg [1:0] read_key;
initial begin scan = 0; read_keys = 1; key1 = 0; … keyN = 0; end
always @(posedge clock)
 begin
    if(read_key == 1) key01 <= k01; ...
    if(read_key == 1) keyN <= kN;
    if(read_key == 1) read_key <= 0;
(1)  if(read_key == 0) scan <= 1;
(1)  if(scan == 1 && key01 == 1) scan <= scan + 1;
(1)  if(scan == 1 && key01 == 0) scan <= 0; ...
(1)  if(scan == N && keyN == 1) scan <= scan + 1;
(2) if(scan == 1 && key01 == 1 && … && keyN == 1) scan <= N + 1;
  end
 end
```

Свойство: `(scan == N + 1 && !(key1 == 0 || … || keyN == 0)) || (scan == 0)`

N	SMV		NuSMV		VCEGAR (2)		SPIN		Алгоритм 4
	состояния	память	состояния	память	состояния	память	состояния	память	состояния
11	494.122	19M	?	76M	39.284	3M	22.519	3M	121
12	1.479.620	63M	?	188M	70.854	4M	45.047	4M	144
13	4.434.197	155M	?	548M	133.187	5M	90.103	5M	169
14	13.295.426	524M	-	-	257.025	7M	180.215	7M	296
20	-	-	-	-	17.318.008	335M	11.534.327	392M	400

Рисунок 5 – Сравнительный анализ для примера 1

Заметим, что число состояний и время работы верификаторов SMV, NuSMV на приведенном примере $O(3^N)$, VCEGAR(2), SPIN $O(2^N)$, тогда как алгоритма 4 – $O(N^2)$. Для первого случая VCEGAR (строки отмечены «1») квадратичный рост наблюдается не только по числу состояний, но и по количеству итераций построения абстракций, так, уже для N=5, число итераций 114, N=6, уже – 219.

Пример 2. Рассмотрим программу на языке Си (формализация на языках Верификаторов опущена):

```
     int max = 4; c = 1; d = 0; z = 1;
cf1: while(true){
cf2:   if(c < max) c := c + 1;
cf3:   if(c > max + d) c := c + z;  };
```

Помимо достижимости переходов, тупиков, ливлоков и недетерминизмов, требуется проверить свойство, что всегда выполняется `c-1 < max`. На рис. 6 представлено множество абстрактных состояний, построенных алгоритмом 4. Атрибут `cf` исполь-зуется в качестве потока управления; повторяющиеся значения вынесены за скобки:

```
          (max=4) X (d=0) X (
   1..3:     cf=(cf1,cf2,cf3) X (c=4);
   4..6:     cf=(cf1,cf2,cf3) X (c=3);
   7..9:     cf=(cf1,cf2,cf3) X (c=2);
   10..11:   cf=(cf1,cf2)     X (c=1) )
```

Рисунок 6 – Абстрактные состояния примера 2

Заметим, что атрибут `z` не входит ни в одно из 11 состояний, построенных алгоритмом 4. Однако, несмотря на недостижимость последнего оператора, методы типа [6], [8], [9], [11], [20] не имеют приемлемого критерия для завершения построения контрпримеров (будут генерироваться предикаты `c > max+d`, `c + z > max+d`, `c + 2z > max+d`,…); при ограничении (`c,d,z = 0..255`) NuSMV выдает ошибку, пытаясь присвоить переменной `c` значение 510; число BDD узлов (и, соответственно, время выполнения) в экспериментах с VCEGAR и SMV заметно сократится (с 106.705 до 9 и с 304.564 до 738 соответственно), если недостижимое присваивание удалить.

4. Проверка темпоральных свойств

Покажем, что метод применим для проверки свойств, заданных формулами линейной темпоральной логики [28].

Определение 11. Истинность LTL формулы φ на состоянии пути p_i записывается $p_i|=\varphi$, определяется индуктивно по структуре формулы φ:

$p_i \ |= \ a \ \equiv \ I(p_i, a)=T$ – *если атомарная формула a истинна в i-м состоянии*;

$p_i \ |= \ \neg\varphi \ \equiv \ p_i \ |\neq \ \varphi$ – *если формула φ не выполняется в p_i*;

$p_i \ |= \ \varphi_1 \mathbf{\&} \varphi_2 \ \equiv \ p_i \ |= \ \varphi_1 \ \& \ p_i \ |= \varphi_2$ – *если в p_i выполняется формула φ_1 и φ_2*;

$p_i \ |= \ \varphi_1 \vee \varphi_2 \ \equiv \ p_i \ |= \ \varphi_1 \ \vee \ p_i \ |= \varphi_2$ – *если в p_i выполняется формула φ_1 или φ_2*;

$p_i \ |= \ \mathbf{X}\varphi \ \equiv \ p_{i+1} \ |= \ \varphi$ – *если φ выполняется в следующем состоянии пути*;

$p_i \ |= \ \varphi_1 \mathbf{U} \varphi_2 \ \equiv \ (\exists k \geq i):(p_k \ |= \ \varphi_2 \ \wedge \ (\forall j: \ i \leq j < k): \ p_j |= \ \varphi_1)$ – *если когда-то в будущем на пути p выполнится φ_2, а до ее выполнения во всех промежуточных состояниях вычисления будет выполняться φ_1.*

Заметим, что добавление семантики темпоральных операторов не противоречит семантике множества `R_SET`. Проверка LTL формул может быть выполнена [29] путем обнаружения сильно связных компонент поведения модели, что, согласно теоремам 1 и 2, обеспечивается алгоритмом 4.

Таким образом, для проверки свойств, заданных формулами LTL, описанный метод нуждается в следующем расширении: 1) функцию `check_property` нужно

задать в виде соответствующей реализации ω-автомата [3], [30]; 2) функции проверки эквивалентности состояний необходимо усилить, добавив состояние автомата, реализующего LTL формулы.

Выводы

Приведены основные алгоритмы и обоснование метода оперативного построения абстракций для автоматической проверки линейных темпоральных свойств. В отличие от [8-11], [20], метод всегда выдает точные результаты, не нуждается в повторных запусках экспериментов для уточнений и не требует построения абстрактных переходов. Метод полностью автоматический, не подразумевает вмешательства со стороны пользователя, как того требуют, например [10], [18], [19]. Отметим также, что метод не чувствителен к связкам атрибутов в постусловиях недостижимых переходов, как, например [4-6], [8-11], [16], [20]. Основным отличием от существующих систем и методов верификации есть то, что абстракции строятся «на лету» в процессе проверки свойств модели. Также необходимо отметить, что описываемый метод не противоречит методам частичного порядка [12], [13] и методам, использующим симметрии [14], [15].

Очевидно, худший случай для метода – модель, в которой для интерпретации переходов и проверки свойств на каждом состоянии пути потребуются все атрибуты. Однако, на практике проверки программ, предусловия переходов, как правило, не большие, а в первом конъюнкте предусловий проверяется атрибут потока управления, отсекая, тем самым, потребность в проверке остальной части предусловий недопустимых переходов.

Литература

1. Режим доступа: http://en.wikipedia.org/wiki/Model_checking
2. Ben-Ari. M. Principles of Spin // Springer Verlag. – 2008. – P. 216.
3. Holzmann G. The model checker SPIN //IEEE transactions on software engineering. –1997. –Vol. 23. – № 5. –P. 279–295.
4. Burch J., Clarke E., McMillan K., Dill D., and Hwang L. Symbolic model checking: 10^20 states and beyond // Information and Computation. – 1992. – Vol. 98, № 2. – P. 142-170.
5. Cimatti A., Clarke E. M., Giunchiglia E., and others. NuSMV 2: An OpenSource Tool for Symbolic Model Checking // In Proceeding of International Conf. on Computer-Aided Verification. – Copenhagen (Denmark). – 2002. – P. 359-364.
6. Jain H., Kroening D., Sharygina N., and Clarke E. VCEGAR: Verilog counterexample guided abstraction refinement // In Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07). – 2007. – P. 4.
7. Jussila T. On Bounded Model Checking of Asynchronous Systems. Doctoral Thesis. Helsinki University of Technology, Laboratory for Theoretical Computer Science. Research report A97. –P. 136. – 2005.
8. Henzinger T., Jhala R., Majumdar R., and Sutre G. Lazy abstraction // In Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. –2002. –Vol. 37. –P. 58–70.
9. Chaki S. A Counterexample Guided Abstraction Refinement Framework for Verifying Concurrent C Programs: Doctoral Thesis. – Carnegie Mellon University. –2005. –P. 253.
10. Clarke E. Model checking and abstraction // In ACM Transactions on programming languages and systems. –1994. –Vol. 16, №5. –P.1512-1542.
11. Pasareanu C., Pelanek R. and Visser W. Predicate abstraction with under-approximation refinement // Logical methods in comp. science. – 2007. – Vol.3. – P. 1-22.
12. Peled D. Combining partial order reductions with on-the-fly model checking // Journal of Formal Methods in System Design. –1996. –Vol.8,№ 1. – P. 39-64.
13. Godefroid P. Partial-order methods for the verification of concurrent systems – an approach to the state-explosion problem // Lecture notes in computer science, Springer-Verlag. –1996. –Vol. 1032. – P. 143.
14. Ip C., Dill D. Better Verification through Symmetry // Formal Methods in System Design. –1996. – Vol. 9. –P. 41-75.

15. Miller A., Donaldson A., and Calder M. Symmetry in temporal logic model checking // ACM Comput. Surv. –2006. –Vol. 38, Issue 3. – P. 37.
16. Lind-Nielsen J., Andersen H., Behrmann G. and oth. Verification of large state/event systems using compositionality and dependency analysis // Journal of Formal Methods in System Design. –2001. – Vol. 18, № 1. – P. 5-23.
17. Lafuente A. Directed Search for the Verification of communication protocols: Doctorial thesis, Institute of computer science, University of Freiburg. – 2003. – P. 157.
18. Bloem R., Ravi K., and Somezi F. Symbolic guided search for CTL model checking // In Design Automation Conference. – 2004. – P. 29-34.
19. Barner S., Glazberg Z., and Rabinovitz I. Wolf-bug hunter for concurrent software using formal methods // In Computer Aided Verification. – 2005. – P.153-157.
20. Clarke E., Grumberg O., Jha S., Lu Y., and Veith H. Counterexample-guided abstraction refinement for symbolic model checking // Journal of the ACM. –2003. –Vol. 50, № 5. – P. 752-794.
21. Колчин А.В. Разработка инструментальных средств для проверки формальных моделей // Проблемы программирования. – К.: Ин-т программных систем НАН Украины. – 2008. – С. 622-626.
22. Керниган Б., Ритчи Д. Язык программирования Си. –2-е изд. – М.: «Вильямс». –2007. – С. 304.
23. Tarjan R. Depth first search and linear graph algorithms // SIAM Journal on Computing. –1972. –Vol. 1, № 2. – P.146-160.
24. Режим доступа: http://spinroot.com/spin/whatispin.html
25. Режим доступа: http://www.kenmcmil.com/smv/linux/
26. Режим доступа: http://nusmv.irst.itc.it/
27. Режим доступа: http://www.cs.cmu.edu/~modelcheck/vcegar/
28. Pnueli A. The temporal logic of programs // Proc. of the 8th IEEE Symposium on Foundation of Computer Science. –1977. –P.46-57.
29. Lichtenstain O., Pnueli A. Checking that finite-state concurrent programs satisfy their linear specification // In Proc. of the 11th ACM Symposium on Principles of Programming Languages. –1984. –P. 97-107.
30. Thomas W. Automata on infinite objects // Handbook of theoretical compute science. –1990. – P. 165-191.

О.В. Колчин

Автоматичний метод оперативної побудови абстракцій при верифікації формальних моделей асинхронних систем

Запропоновано метод побудови точних абстракцій «на льоту» та його використання у верифікації формальних моделей. Метод заснований на тому, що кожний пройдений стан моделі характеризується неповним набором атрибутів, при цьому досягається суттєве скорочення числа станів необхідних для аналізу моделі, що верифікується. Описано основні алгоритми побудови абстракцій, ефективність використання проілюстровано на прикладах. Наведено необхідні розширення для перевірки темпоральних властивостей.

A. Kolchin

An Automatic Method for On-The-Fly Abstractions Building During Model Verification of Asynchronous Systems

A method for “on-the-fly” exact abstraction construction for model checking is proposed. The basis of the method is storing of incomplete set of attributes in visited states. Due to this fact the number of needed for model analysis states is substantially smaller. The main algorithms for abstractions building are described. Effectiveness of the method applying is demonstrated with examples. Enhancements needed for temporal properties verification are described.

Статья поступила в редакцию 21.07.2008.